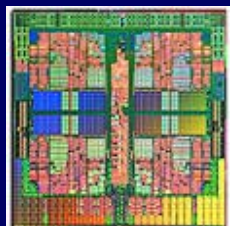
Parallel Computing with GPUs

David Kaeli
Perhaad Mistry
Rodrigo Dominguez
Dana Schaa
Matthew Sellitto

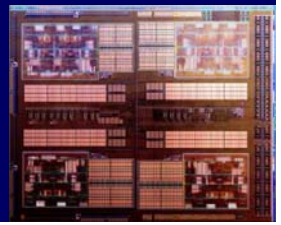
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA



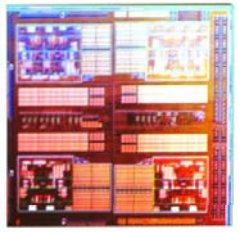
Northeastern

Objectives of this class

- Develop a new appreciation for thinking in parallel
- Provide an overview of many-core and parallel computing
- Present recent advances in GPU computing in terms of both hardware and software
- Provide programming exercises and hands-on experience writing GPU programs
- Learn the basics of pthreads, CUDA and OpenCL
- Discuss recent hardware advances by NVIDIA and AMD

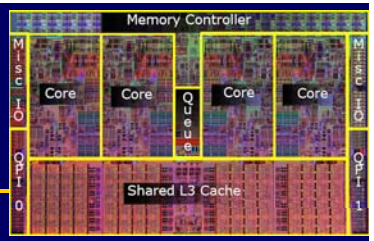


Course Schedule



- April 7 - Parallel hardware and software - control and data flow, shared memory vs. message passing, thread-level vs. data-level parallelism, pthreads programming
- April 14 - Introduction to GPU Computing with CUDA Topics - GPU programming, NVIDIA hardware
- April 28 - Intermediate CUDA Programming Topics - Syntax, sample programs, debugging
- May 5 - Introduction to OpenCL Computing Topics - Syntax, examples
- May 12 - OpenCL programming on AMD GPUs and X86 CPUs Topics - Hands-on programming and debugging

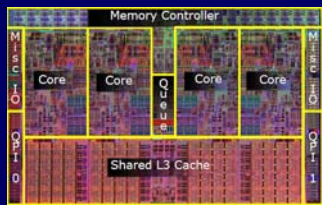
Today's Class



- Parallel hardware and software
  - Multi-core CPUs
  - SIMD
  - Vectors
  - GPUs
  - Programming models
  - Thread-based programming with pThreads
- Concepts
  - Control and data flow
  - Shared memory vs. message passing
  - Thread-level vs. data-level parallelism

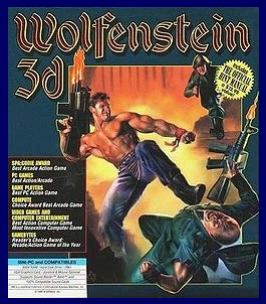
The move to multi-core computing

- The CPU industry has elected to jump off the cycle-time scaling bandwagon
  - Power/thermal constraints have become a limiting factor
  - Clock speeds have not changed
  - The memory wall persists and multi-core places further pressure on this problem
- Microprocessor manufacturers are producing high-volume CPUs with 8-16 cores on-chip
  - AMD Bulldozer has up to 16 cores
  - SIMD/vector extensions – SSE (streaming SIMD extensions) and AVX (advanced vector extensions)
  - Also seeing multi-core in the embedded domain



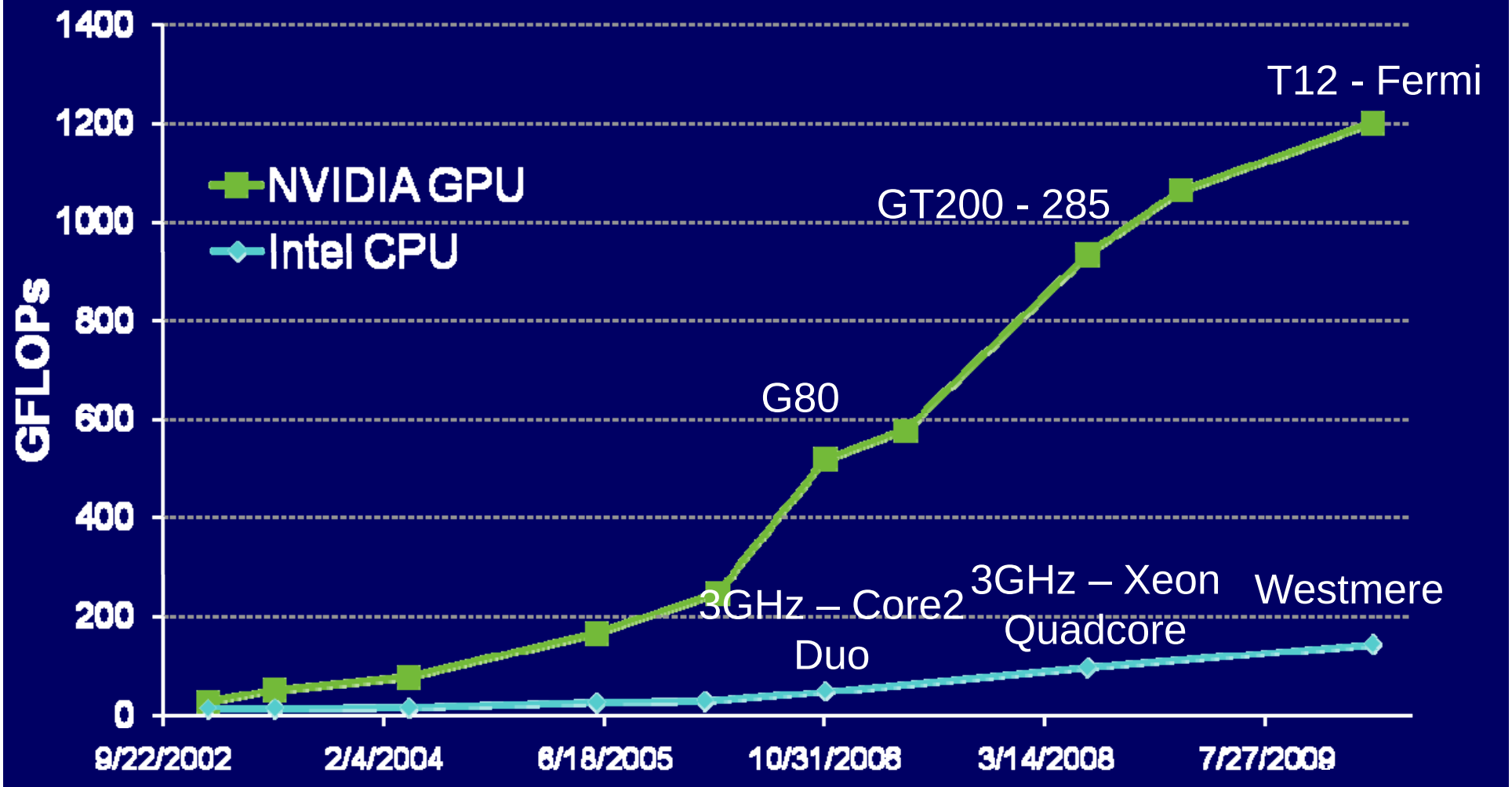
Why are Graphics Processors of interest?

- **Graphics Processing Units**
  - More than 65% of Americans played a video game in 2009
  - High-end - primarily used for 3-D rendering for videogame graphics and movie animation
  - Mid/low-end – primarily used for computer displays
  - Manufacturers include NVIDIA, AMD/ATI, IBM-Cell
  - Very competitive commodities market



Northeastern

NVIDIA Comparison of CPU and GPU Hardware Architectures (SP GFLOPS)



Source: NVIDIA

Northeastern

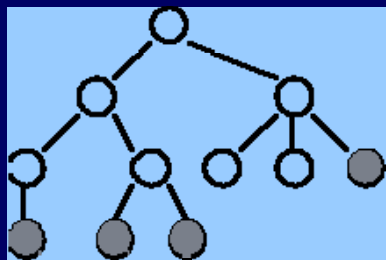
Comparison of CPU and GPU Hardware Architectures

CPU/GPU	Single precision TFLOPs	Cores	GFLOPs/ Watt	$/GFLOP
NVIDIA 285	1.06	240	5.8	$0.09
NVIDIA 295	1.79	480	6.2	$0.08
NVIDIA 480	1.34	480	5.4	$0.41
AMD HD 6990	5.10	3072	11.3	$0.14
AMD HD 5870	2.72	1600	14.5	$0.16
AMD HD 4890	1.36	800	7.2	$0.18
Intel I-7 965	0.051	4	0.39	$11.02

Source: NVIDIA, AMD and Intel

And parallel software?

- Hardware vendors have tried to roll their own
  - NVIDIA's CUDA
  - AMD's CTM and Brook+
  - Intel Ct Technology
- Software vendors are developing new parallelization technology
  - Multi-core aware operating systems – Microsoft (BarrelFish), Apple (Grand Central Dispatch), others
  - Parallelizing compilers – Microsoft Visual, LLVM, Open64, Portland Group, IBM XLC, others
  - Portable frameworks for heterogeneous computing – Kronos/OpenCL
  - Hybrids – AccelerEyes (Matlab)

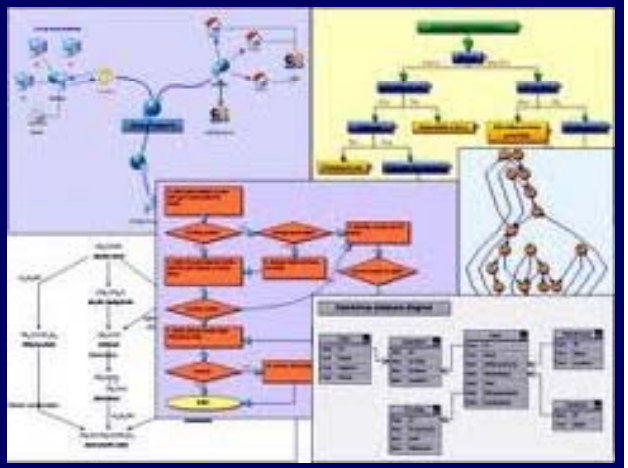


So how to start writing parallel programs?

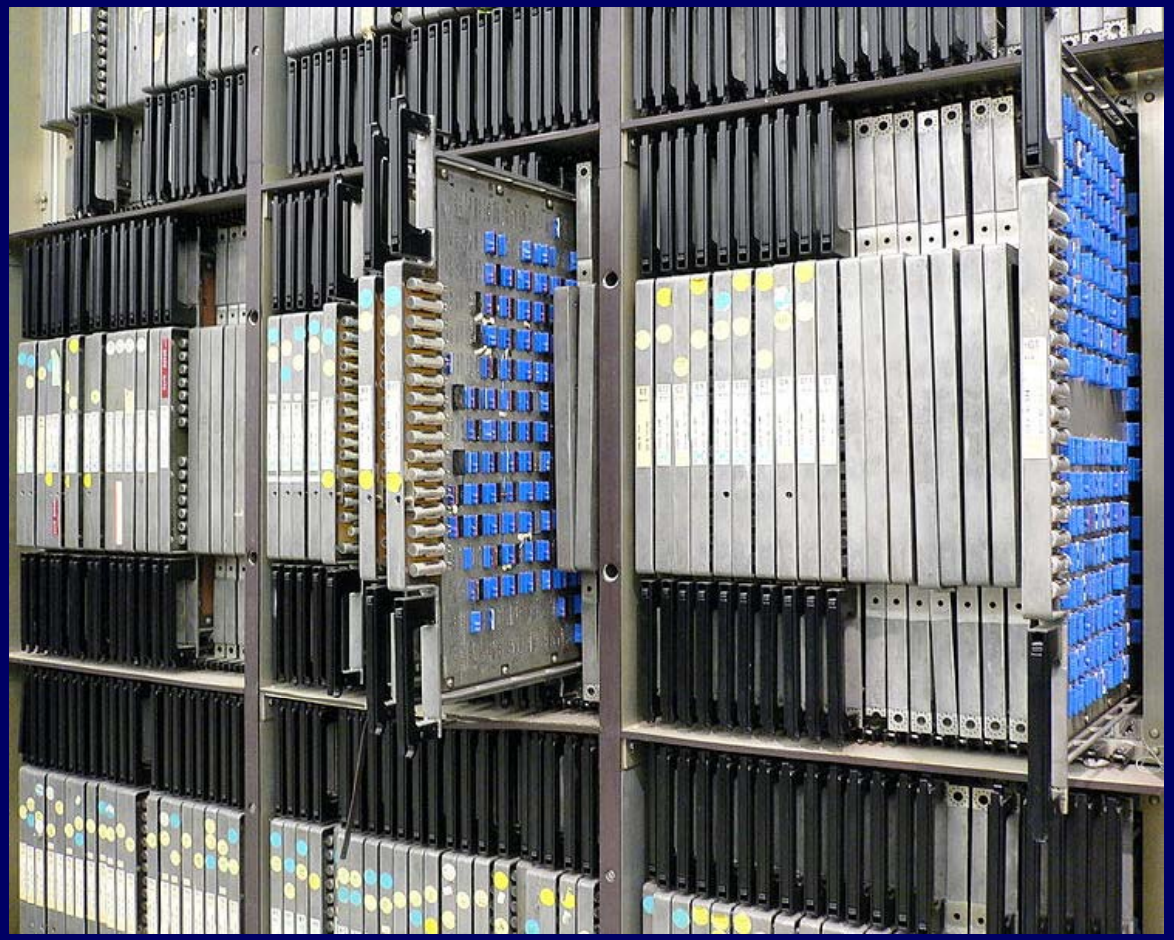
- If we use a shared memory system model
  - A single contiguous memory address space
  - Hardware/architecture provide for memory coherency and consistency
  - Threading can be used to manage synchronization and serialization where needed
  - Significantly reduces the burden of parallelizing programs
- If we use a distributed systems model
  - Multiple memory address spaces are available
  - Coherency and synchronization across these spaces is managed by explicit (versus implicit) synchronization
  - The programmer has the task of issuing explicit communication commands to manage synchronization of tasks and distributed memory

Issues with Control Flow

- Imagine a world without control flow?
  - Parsing is heavily control dependent
  - Many desktop applications are dominated by control flow
- We would only have computation without any interpretation of the answers
- Control flow significantly impacts our ability to carry out operations in parallel
- Can we avoid control flow?



Reasoning about Parallelism



We need to start thinking parallel

- To begin to utilize parallel systems such as multi-core CPUs, many-core GPUs and clusters, you need to understand parallelism
- We will explore this through some simple exercises
- We will explore some questions that will help you understand the challenges we must overcome to exploit parallel resources

Let's Bake Some Cakes

- You are trying to bake 3 blueberry pound cakes
- Cake ingredients are as follows:

  - 1 cup butter, softened
  - 1 cup sugar
  - 4 large eggs
  - 1 teaspoon vanilla extract
  - 1/2 teaspoon salt
  - 1/4 teaspoon nutmeg
  - 1 1/2 cups flour 1 cup blueberries

Cake Baking 101

The recipe for a single cake is as follows:

- Step 1: Preheat oven to 325˚ F (160˚ C). Grease and flour your cake pan.
- Step 2: In large bowl, beat together with a mixer butter and sugar at medium speed until light and fluffy. Add eggs, vanilla, salt and nutmeg. Beat until thoroughly blended. Reduce mixer speed to low and add flour, 1/2 cup at a time, beating just until blended.
- Step 3: Gently fold in blueberries. Spread evenly in prepared baking pan. Bake for 35 minutes.
- Step 4: Allow to cool for 20 minutes in pan. Remove from pan

Cake Baking 101

- Your task is to cook 3 cakes as efficiently as possible
- Assume that you only have one oven (large enough to hold one cake), one large bowl, one cake pan, and one mixer
- Come up with a schedule to make three cakes as quickly as possible
- Identify the bottlenecks in completing this task

Cake Baking 101

- Assume now that you have three bowls, 3 cake pans and 3 mixers. How much faster is the process now that you have additional resources?

Northeastern

Cake Baking 101

- Assume now that you have two friends that will help you cook, and that you have three ovens that can accommodate all three cakes.
- How will this change the schedule you arrived at in first scenario?

Northeastern

Back to Cake Baking 101

- Compare the cake-making task to computing 3 iterations of a loop on a parallel computer
- Identify data-level parallelism and task-level parallelism in the cake-making loop

Cost/Performance

- Next, consider the following costs associated with cake baking
  - Ovens = $200/oven
  - Cake pans = $10/pan
  - Mixers = $25/mixer
  - Cooks = $75/cook
  - Bowls = $5/bowl
- Consider the following latencies
  - Grease and flour pan = 5 minutes
  - Mixing time = 40 minutes
  - Cooking time = 30 minutes
  - Cooling time = 20 minutes
- Find the best cost vs. performance given these objectives:
  - Performance in cakes/minute
  - Performance in cakes /$

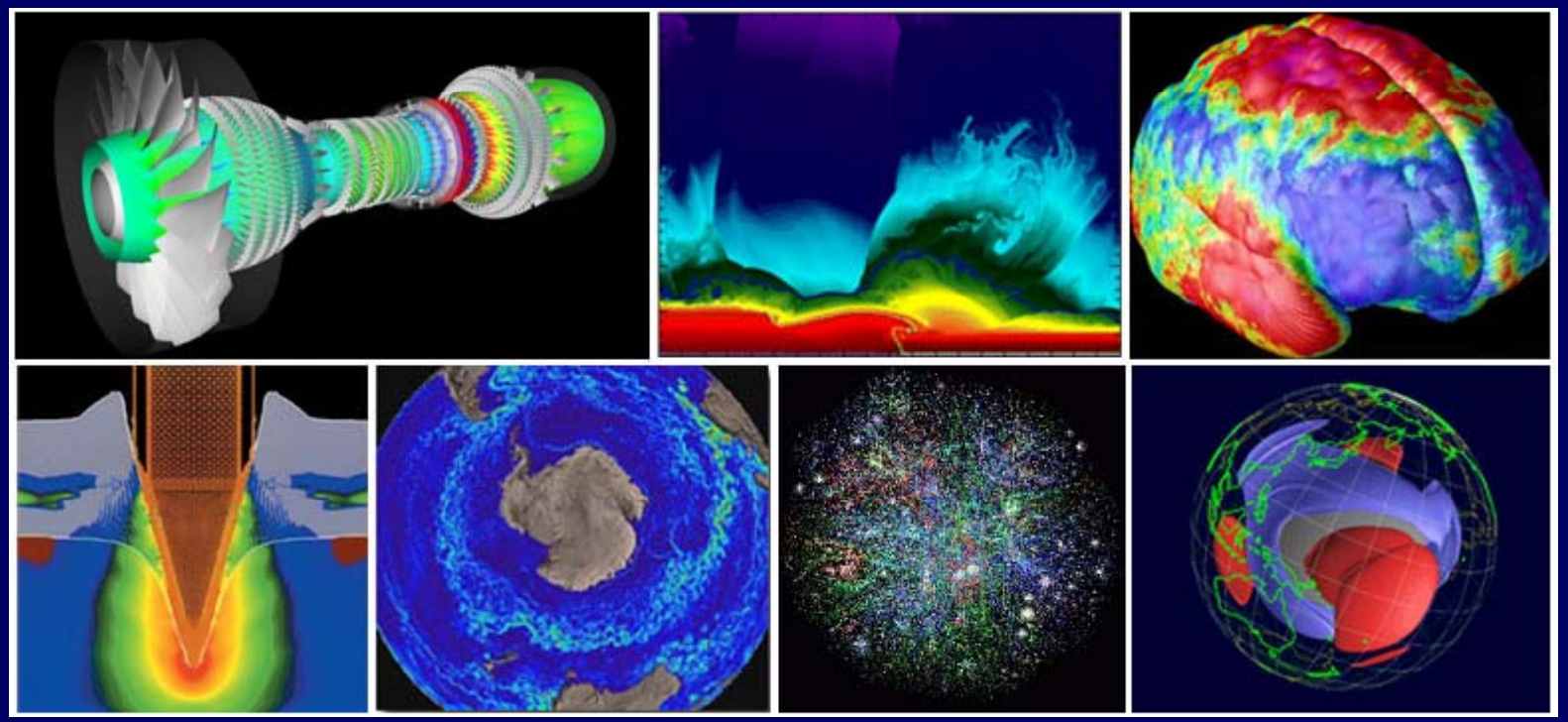
Parallelizing our lives

- Many of the tasks we perform in our everyday lives include significant parallelism
- Can you name some of these?
- Write down a list of your daily activities (e.g., shower, get dressed, eat breakfast, dry your hair)
  - Identify at least 12-15 activities
- Consider which of these activities could be carried out concurrently
  - Identify pairs of parallelizable activities

What is wrong with our world?

- Next, consider why many of these cannot presently be carried out in parallel
  - What would need to be changed in our physical world (e.g., showers, cars, Ipods) to allow us to complete many of these activities in parallel
  - How often is parallelism inhibited by our inability of carrying out two things at the same time?
- Estimate how much more quickly it would take to carry out these activities if you could change these physical systems

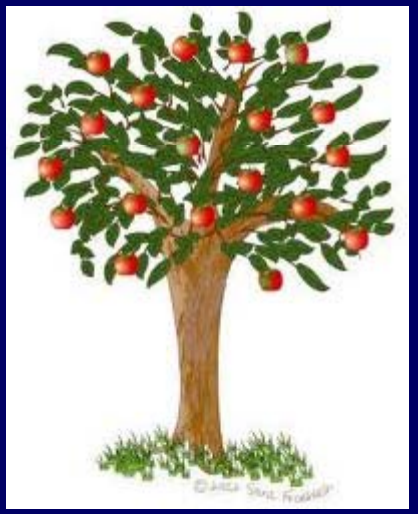
What is wrong with our world? Nothing!!



There is rampant parallelism in the natural world!

Let’s pick some apples

If I want to pick all the apples, but have only one picker and one ladder, how will I pick all the apples?



Northeastern

Let’s pick some apples

What happens if I get a second picker (a shorter picker) but keep one ladder, how will I pick all the apples?

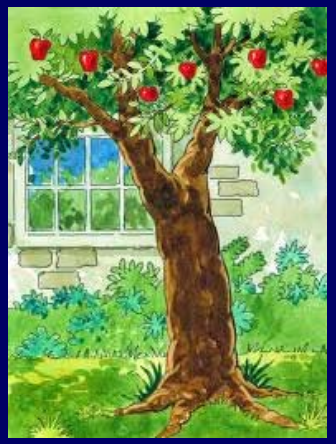
Let’s pick some apples

What happens if I get a second picker and a second ladder, how will I pick all the apples?

Northeastern

Let's pick some apples – heterogeneity…

What changes if the trees have a very different number of apples on them?



Northeastern

Parallel processing terminology

- Task
  - A logically discrete section of computational work
  - A task is typically a program or program-like set of instructions that is executed by a processor
- Serial Execution
  - Execution of a program one statement at a time
  - Virtually all parallel tasks will have sections of a parallel program that must be executed serially
- Parallel Execution
  - Execution of a program by more than one task, with each task being able to execute the same or different statements at the same moment in time
- Communications
  - Parallel tasks typically need to exchange data through a shared memory bus or over a network

Northeastern

Parallel processing terminology

- Granularity
  - A qualitative measure of the ratio of computation to communication
  - *Coarse:* relatively large amounts of computational work are done between communication events
  - *Fine:* relatively small amounts of computational work are done between communication events
- Parallel Overhead
  - The amount of time required to coordinate parallel tasks, as opposed to doing useful work
  - Synchronizations and data communication
  - Middleware execution overhead
- Massively Parallel
  - Hardware that comprises a given parallel system - having many processors
  - The meaning of "many" keeps increasing
- Embarrassingly Parallel
  - Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks ☺

Parallel processing terminology

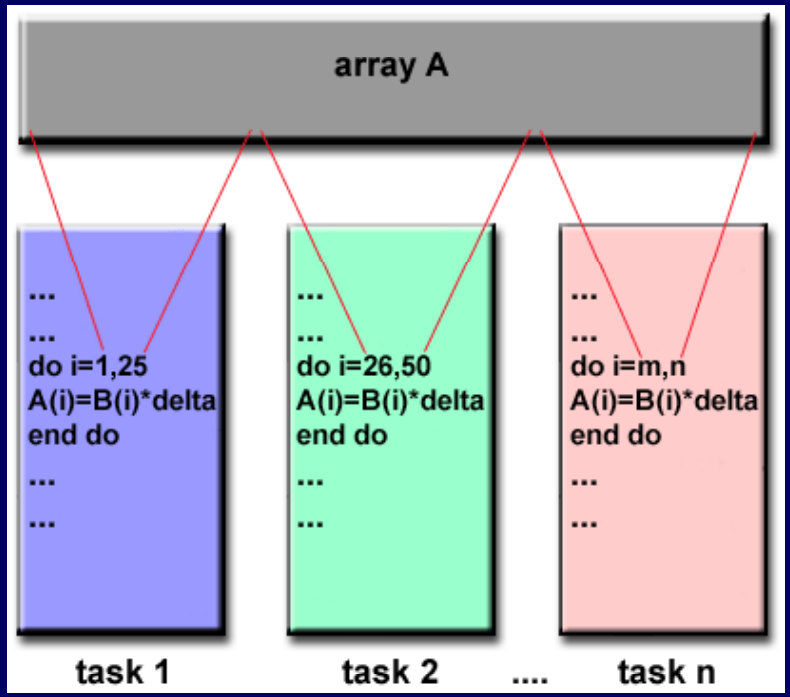
- Threads Model
  - In parallel programming, a single process can have multiple, concurrent execution paths
  - The main program performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run concurrently
  - Each thread has local data, but also shares the entire resources of main program
  - Saves the overhead associated with replicating a program's resources for each thread
  - Each thread also benefits from a global memory view because it shares the memory space of the main program

Parallel processing terminology

- Types of Synchronization:
  - Barrier
    - Usually implies that all tasks are involved
    - Each task performs its work until it reaches the barrier. It then stops, or "blocks"
    - When the last task reaches the barrier, all tasks are synchronized
  - Lock / semaphore
    - Can involve any number of tasks
    - Typically used to serialize (protect) access to global data or a section of code
    - The first task to acquire the lock "sets" it - this task can then safely (serially) access the protected data or code

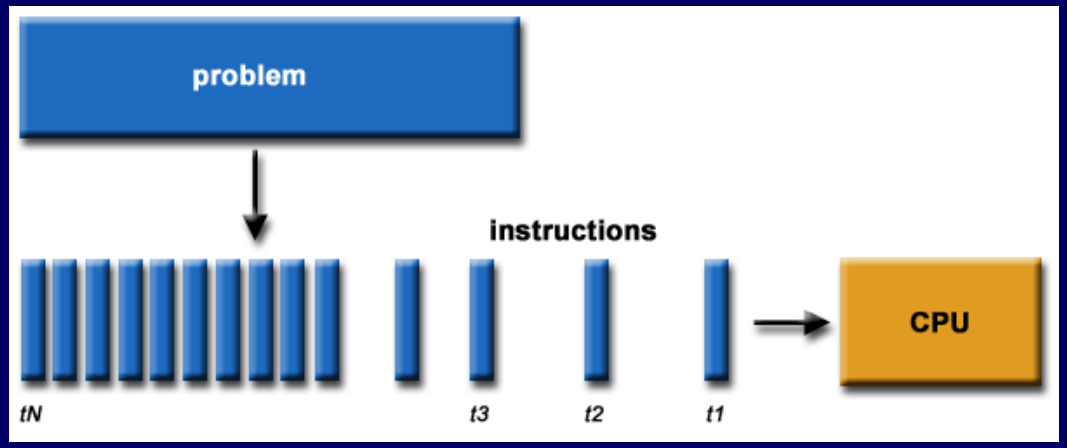
Parallel processing terminology

- Data Parallelism
  - Most parallel execution operates on a data set
  - The data set is typically organized as an array or multi-dimensional matrix
  - A set of tasks work collectively on the same data structure
  - Each task/thread works on a different partition of the same data structure
  - Tasks perform the same operation on their partition of work, for example, “multiply every array element by delta"



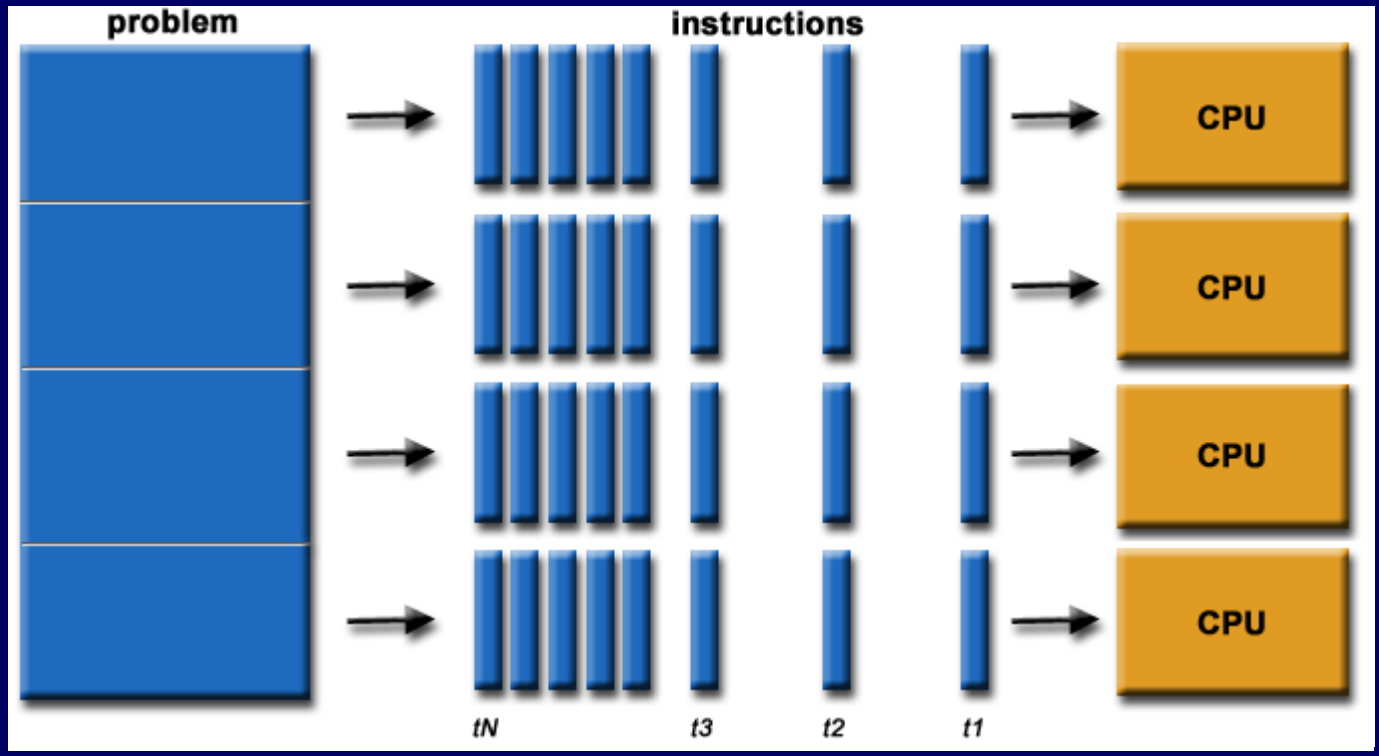
The grain of computation

- Programs can be decomposed into:
  - Processes or Threads
  - Functions
  - Kernels
  - Loops
  - Basic blocks
  - Instructions
- CPUs are designed using pipelining



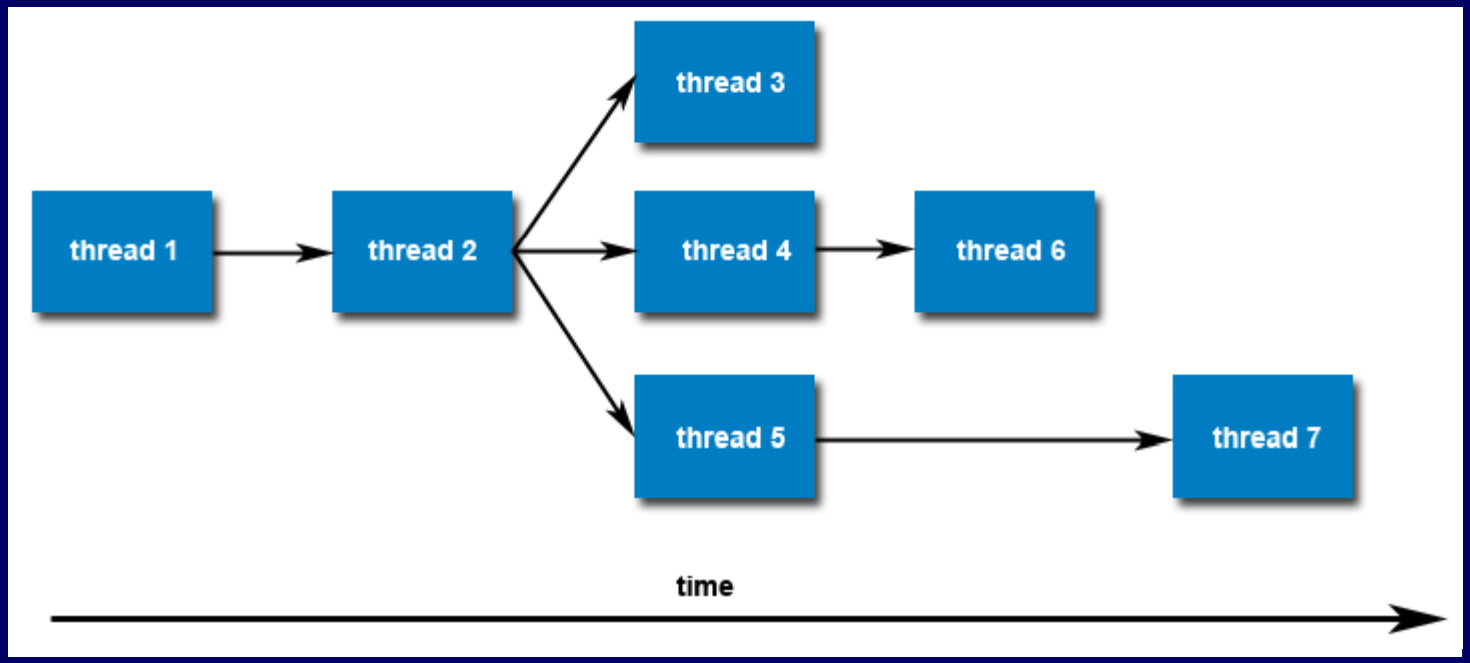
Decomposing computation

- Programs can be decomposed into parallel subproblems:



Decomposing computation

- Single programs can be further decomposed into parallel subproblems using threads:



What is a thread?

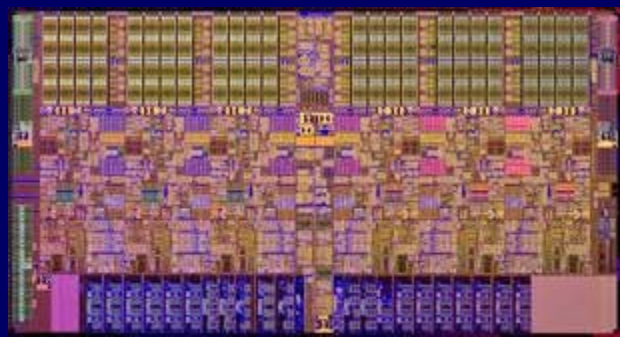
- Process:
  - An address space with 1 or more threads executing within that address space, and the required system resources for those threads
  - A *program* that is running
- Thread:
  - A sequence of control within a process
  - Shares the resources of the process

Advantages and Drawbacks of Threads

- Advantages:
  - The overhead for creating a thread is significantly less than that for creating a process
  - Multitasking, wherein one process serves multiple clients
  - Switching between threads requires the OS to do much less work than switching between processes – lightweight
  - Hardware can be designed to further reduce this overhead

Advantages and Drawbacks of Threads

- Drawbacks:
  - Not as widely available on all platforms
  - Writing multithreaded programs requires more careful thought
  - More difficult to debug than single threaded programs
  - For single processor/core machines, creating several threads in a program may not necessarily produce an increase in performance (the overhead of thread management may dominate)



POSIX Threads (pthreads)

- IEEE's POSIX Threads Model:
  - Programming models for threads on a UNIX platform
  - pthreads are included in the international standards ISO/IEC9945-1
- pthreads programming model:
  - Creation of threads
  - Managing thread execution
  - Managing the shared resources of the process

Pthreads – The basics: Main thread

- Initial thread created when main() (in C) or PROGRAM (in Fortran) are invoked by the process loader
- Once in main(), the application has the ability to create daughter threads
- If the main thread returns, the process terminates even if there are running threads in that process, unless special precautions are taken
- To explicitly avoid terminating the entire process, use pthread_exit()

Northeastern

Pthreads – The basics

- Thread termination methods:
  - Implicit termination:
    - thread function execution is completed
  - Explicit termination:
    - calling `pthread_exit()` within the thread
    - calling `pthread_cancel()` to terminate other threads
- For numerically intensive routines, it is suggested that the application calls *p* threads if there are *p* available processors
- The program in C++ calls the pthread.h header file
- Pthreads related statements are preceded by the `pthread_` prefix (except for semaphores)

Pthreads – hello.cpp

```
1.  //************************************************************
2.  //     This is a sample threaded program in C++.  The main thread creates
3.  //     4 daughter threads.  Each child thread simply prints out a message
4.  //  before exiting.  Notice that we have set the thread attributes to joinable and
5.  //     of system scope.
6.  //************************************************************
7.  #include <iostream.h>
8.  #include <stdio.h>
9.  #include <pthread.h>
10.
11.  #define NUM_THREADS 4
12.
13.  void *thread_function( void *arg );
14.
15.  int main( void )
16.  {
17.      int i, tmp;
18.      int arg[NUM_THREADS] = {0,1,2,3};
19.
20.      pthread_t thread[NUM_THREADS];
21.      pthread_attr_t attr;
22.
23.      // initialize and set the thread attributes
24.      pthread_attr_init( &attr );
25.      pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_JOINABLE );
26.      pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
27.
```

Northeastern

Pthreads – hello.cpp

```
28        // creating threads
29        for ( i=0; i<NUM_THREADS; i++ )
30        {
31           tmp = pthread_create( &thread[i], &attr, thread_function, (void
      *)&arg[i] );
32
33           if ( tmp != 0 )
34           {
35              cout << "Creating thread " << i << " failed!" << endl;
36              return 1;
37           }
38        }
39
40        // joining threads
41        for ( i=0; i<NUM_THREADS; i++ )
42        {
43           tmp = pthread_join( thread[i], NULL );
44           if ( tmp != 0 )
45           {
46              cout << "Joining thread " << i << " failed!" << endl;
47              return 1;
48           }
49        }
50
51        return 0;
52     }
53
```

Northeastern

Pthreads – hello.cpp

```
54    //*************************************************************
55    //   This is the function each thread is going to run.  It simply asks
56    //   the thread to print out a message.  Notice the pointer acrobatics.
57    //*************************************************************
58    void *thread_function( void *arg )
59    {
60      int id;
61
62      id = *((int *)arg);
63
64      printf( "Hello from thread %d!\n", id );
65      pthread_exit( NULL );
66    }
```

Northeastern

Discussion on hello.cpp

- How to compile:
  - On our Redhat Linux system, use:

```
g++ -pthread filename.cpp –o filename
```

- Creating a thread:

```
int pthread_create( pthread_t *thread, pthread_attr_t *attr,
  void *(*thread_function)(void *), void *arg );
```

  - First argument – pointer to the identifier of the created thread
  - Second argument – thread attributes
  - Third argument – pointer to the function the thread will execute
  - Fourth argument – the argument of the executed function (usually a struct)
  - Returns 0 for success

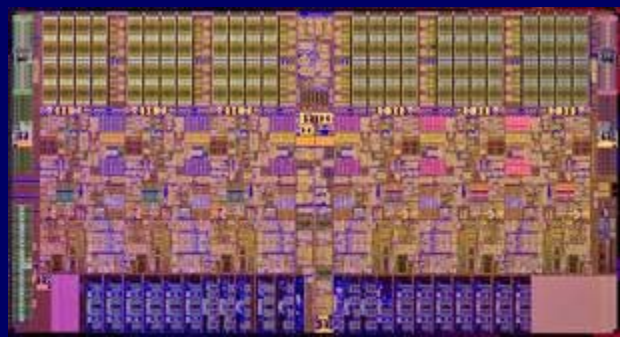
Discussion on hello.cpp

- Waiting for the threads to finish:

```
int pthread_join(pthread_t thread, void **thread_return)
```

  - Main thread will wait for daughter thread *`thread`* to finish
  - First argument – the thread to wait for
  - Second argument – pointer to a pointer to the return value from the thread
  - Returns 0 for success
  - Threads should always be joined; otherwise, a thread might keep on running even when the main thread has already terminated

- Compile and run the example

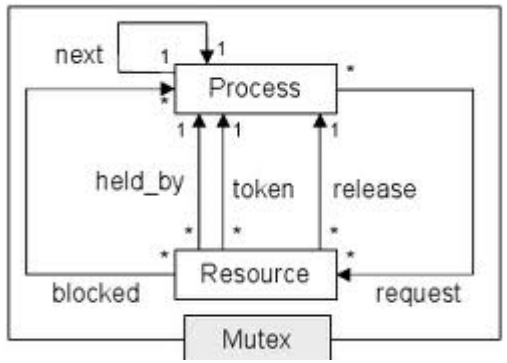


Threads Programming Models

- Pipeline model – threads are run one after the other
- Master-slave model – master (main) thread doesn't do any work, it just waits for the slave threads to finish working
- Equal-worker model – all threads do the same work

Thread Synchronization Mechanisms

- Mutual exclusion (mutex):
  - Guards against multiple threads modifying the same shared data simultaneously
  - Provides locking/unlocking critical code sections where shared data is modified
  - Each thread waits for the mutex to be unlocked (by the thread who locked it) before performing the code section



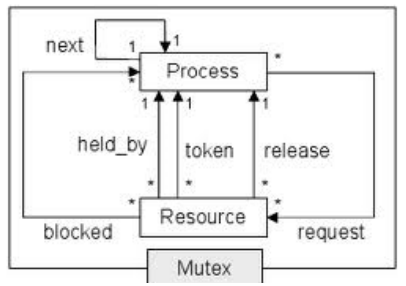
Northeastern

Thread Synchronization Mechanisms

- Basic Mutex Functions:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
  pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  - A new data type named `pthread_mutex_t` is designated for mutexes
  - A mutex is like a key (to access the code section) that is handed to only one thread at a time
  - The attribute of a mutex can be controlled by using the `pthread_mutex_init()`function
  - The lock/unlock functions work in tandem



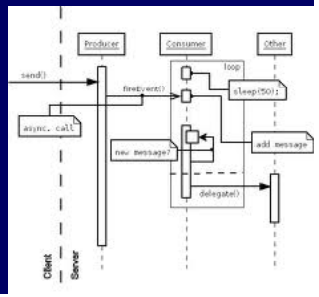
Thread Synchronization Mechanisms

```
#include <pthread.h>
pthread_mutex_t my_mutex;     // should be of global scope
...
int main()
{
   int tmp;
   ...
   // initialize the mutex
   tmp = pthread_mutex_init( &my_mutex, NULL );
   ...
   // create threads
   ...
   pthread_mutex_lock( &my_mutex );
        do_something_private();
   pthread_mutex_unlock( &my_mutex );
         ...
   return 0;
}
```

- Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If locked, it waits until it is unlocked. Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

Thread Synchronization Mechanisms

- Consider the code in the mutex example
- One thread increments the shared variable (shared_target) and the other decrements
- Illustrates many of the basics for managing shared data
- Experiment with the number of threads and the number of iterations
- What is the relationship between these parms?



Thread Synchronization Mechanisms

- Counting Semaphores:
  - Permit a limited number of threads to execute a section of the code
  - Similar to mutexes
  - Should include the `semaphore.h` header file
  - Semaphore functions do not have `pthread_` prefixes; instead, they have `sem_` prefixes

Thread Synchronization Mechanisms

- Basic Semaphore Functions:
  - Creating a semaphore:

```
int sem_init(sem_t *sem, int pshared, unsigned int
  value);
```

    - initializes a semaphore object pointed to by `sem`
    - `pshared` is a sharing option; a value of *0* means the semaphore is local to the calling process
    - gives an initial value `value` to the semaphore
  - Terminating a semaphore:

```
int sem_destroy(sem_t *sem);
```

    - frees the resources allocated to the semaphore `sem`
    - usually called after `pthread_join()`
    - an error will occur if a semaphore is destroyed for which a thread is waiting

Northeastern

Thread Synchronization Mechanisms

- Semaphore control:

```
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

  - `sem_post` *atomically* increases the value of a semaphore by 1, i.e., when 2 threads call `sem_post` simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)
  - `sem_wait` *atomically* decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

Thread Synchronization Mechanisms

```
#include <pthread.h>
#include <semaphore.h>
void *thread_function( void *arg );
...
sem_t semaphore;        // also a global variable just like mutexes
...
int main()
{
    int tmp;
    ...
      // initialize the semaphore
    tmp = sem_init( &semaphore, 0, 0 );
    ...
    // create threads
    pthread_create( &thread[i], NULL, thread_function, NULL );
    ...
    while ( still_has_something_to_do() )
    {
      sem_post( &semaphore );
            ...
    }
    ...
    pthread_join( thread[i], NULL );
    sem_destroy( &semaphore );
    return 0;
}
```

Thread Synchronization Mechanisms

```
void *thread_function( void *arg )
{
   sem_wait( &semaphore );
   perform_task_when_sem_open();
   ...
   pthread_exit( NULL );
}
```

- The main thread increments the semaphore's count value in the while loop
- The threads wait until the semaphore's count value is non-zero before performing `perform_task_when_sem_open()`
- Daughter thread activities stop only when `pthread_join()`is called

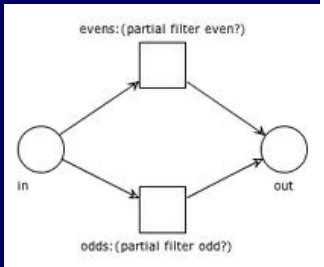
Northeastern

Thread Synchronization Mechanisms

- Look through the semaphore code provided
- The application performs a simple simulation of a producer/consumer: producing and buying milk
- Experiment with increasing the number of threads

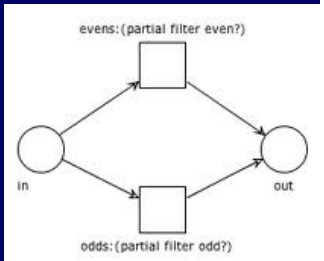
Condition Variables

- Used for communicating information about the state of shared data
- Can make the execution of sections of a code by a thread depend on the state of a data structure or another running thread
- Condition variables are used for signaling, not for mutual exclusion; a mutex is needed to synchronize access to shared data



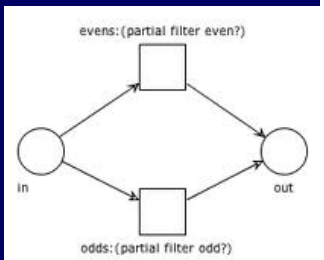
Another pthreads example

- Run the pthreads matmul.cpp program
- Experiment with its performance
- Compile and run the serial pi program (make sure to pass an input parameter – an integer)
- Now attempt to parallelize it with pthreads! (homework)



A parallel implementation of pi.cpp

```
#include <stdio.h>
#include <pthread.h>
int n, num_threads;
double d, pi;
pthread_mutex_t reduction_mutex;
pthread_t *tid;
void *PIworker(void *arg) { int i, myid;
  double s, x, mypi; myid = *(int *)arg;
s = 0.0;
for (i=myid+1; i<=n; i+=num_threads)
   { x = (i-0.5)*d; s += 4.0/(1.0+x*x);
    }
   mypi = d*s;
   pthread_mutex_lock(&reduction_mutex);
   pi += mypi;
   pthread_mutex_unlock(&reduction_mutex);
```



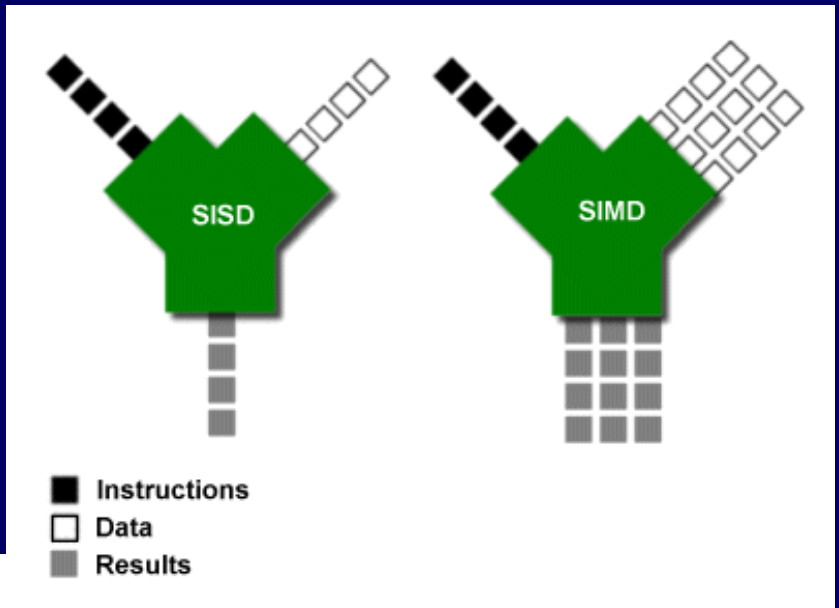
A parallel implementation of pi.cpp

```
pthread_exit(0);
}
main(int argc, char **argv) { int i; int *id; n =
  atoi(argv[1]);
  num_threads = atoi(argv[2]);
  d = 1.0/n;
  pi = 0.0;
  id = (int *) calloc(n,sizeof(int)); t
  id = (pthread_t *) calloc(num_threads,
  sizeof(pthread_t));
  if(pthread_mutex_init(&reduction_mutex,NULL))
     { fprintf(stderr, "Cannot init lock\n"); exit(0); };
  for (i=0; i<num+_threads; i++)
   { id[i] = i;
   if(pthread_create(&tid[i],NULL, PIworker,(void
   *)&id[i])) { exit(1);
    };
  };
  for (i=0; i<num_threads; i++)
  pthread_join(tid[i],NULL);
  printf("pi=%.15f\n", pi); }
```

Northeastern

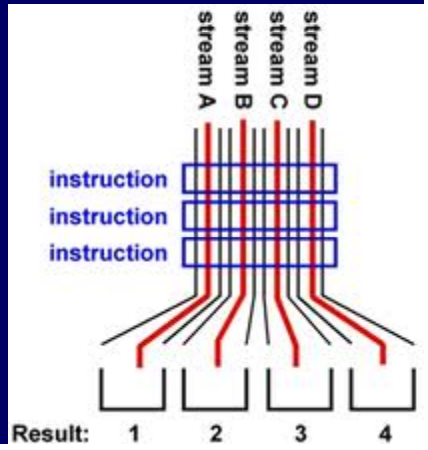
SIMD Execution

- Single Instruction Multiple Data [Flynn]
- Also referred to as vectorization
- Effectively used for over 35 years to exploit data-level parallelism (CDC Star100 and ILLIAC-IV)
- Execute the same instruction on different data
- X86 extensions MMX, SSE, 3DNow, AVX
- AltiVEC (PowerPC), Vis (SPARC)



SIMD Execution – Advantages

- Exploits rampant data parallelism
- Better code density
- Lower code decoding overhead
- Potentially better memory efficiency – implicit data locality
- Efficient for streaming and multimedia applications
- Can effectively handle irregular data patterns (e.g., swizzles)



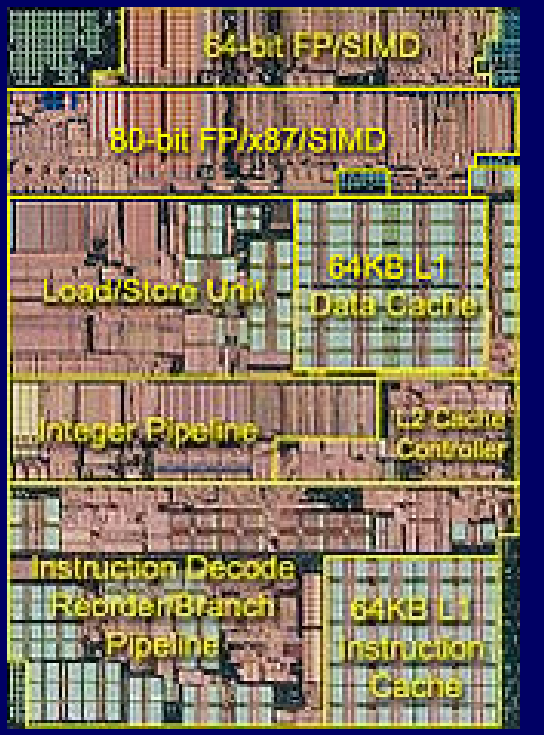
SIMD Execution – Disadvantages

- Not all algorithms can be vectorized
- Gathering data into SIMD registers and scattering it to the correct destination locations is tricky and can be inefficient (swizzles start to address this issue)
- Specific instructions like rotations and three-operand adds are uncommon in SIMD instruction sets
- Instruction sets are architecture-specific: old processors and non-x86 processors lack SSE entirely - programmers must provide non-vectorized implementations (or different vectorized implementations) for them
- The early MMX instruction set shared a register file with the floating-point stack, which caused inefficiencies when mixing floating-point and MMX code

Northeastern

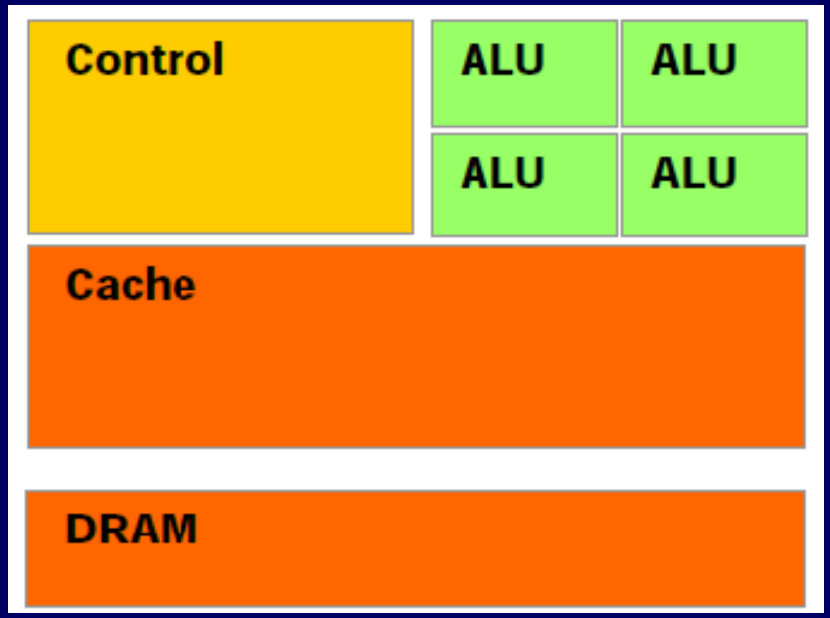
SIMD Execution – SSE

- Supported by most C/C++/Fortran compilers
- On our Linux system, use the –msse switch
- Using pi serial code, experiment with using this switch
- Increase the number of loops to get timing statistics

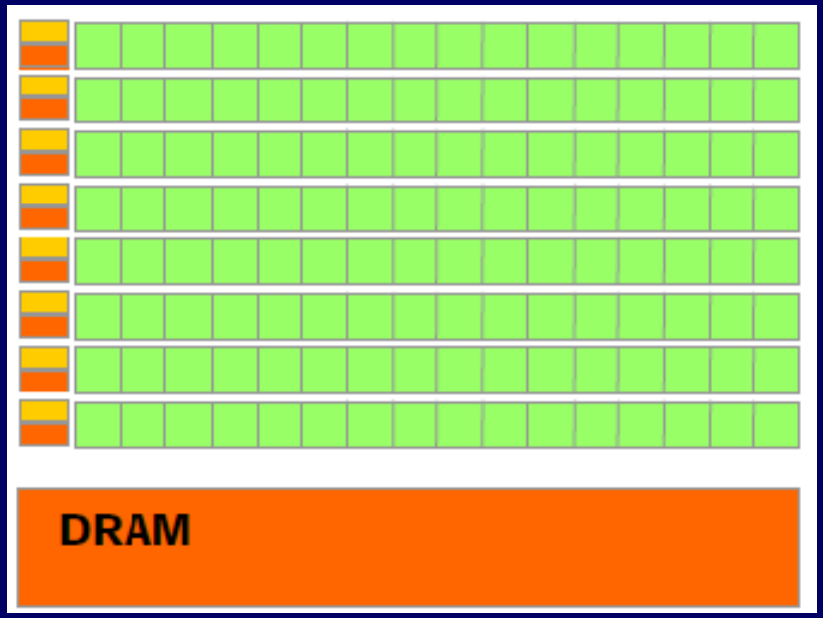


Comparison of CPU and GPU Hardware Architectures

CPU: Cache heavy, focused on individual thread performance



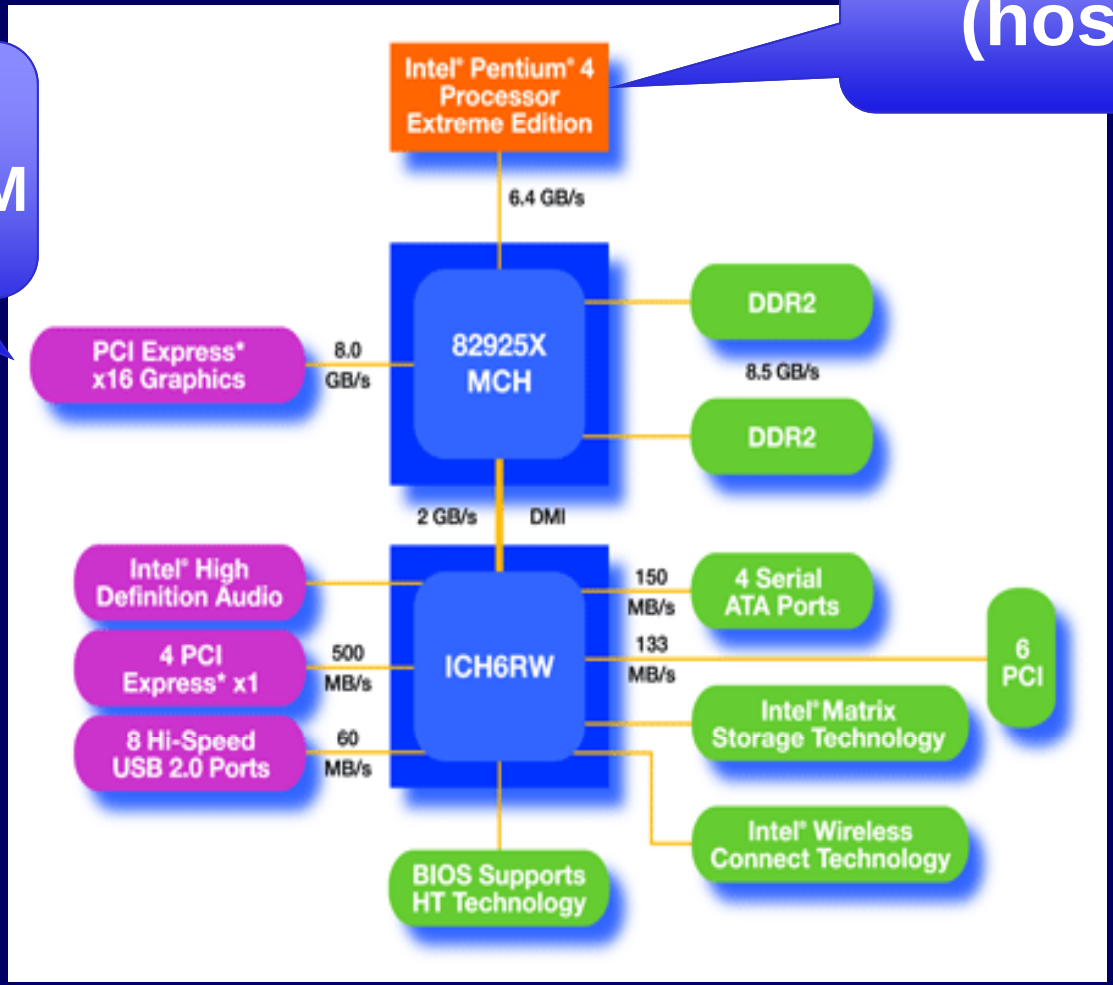
GPU: ALU heavy, massively parallel, throughput-oriented



Traditional CPU/GPU Relationship

CPU (host)

GPU w/ local DRAM (device)



A wide range of GPU applications

- 3D image analysis
- Adaptive radiation therapy
- Acoustics
- Astronomy
- Audio
- Automobile vision
- Bioinfomatics
- Biological simulation
- Broadcast
- Cellular automata
- Fluid dynamics
- Computer vision
- Cryptography
- CT reconstruction
- Data mining
- Digital cinema / projections
- Electromagnetic simulation
- Equity trading
- Film
- Financial
- GIS
- Holographics cinema
- Intrusion detection
- Machine learning
- Mathematics research
- Military
- Mine planning
- Molecular dynamics
- MRI reconstruction
- Multispectral imaging
- N-body simulation
- Network processing
- Neural network
- Oceanographic research
- Optical inspection
- Particle physics
- Protein folding
- Quantum chemistry
- Ray tracing
- Radar
- Reservoir simulation
- Robotic vision / AI
- Robotic surgery
- Satellite data analysis
- Seismic imaging
- Surgery simulation
- Surveillance
- Ultrasound
- Video conferencing
- Telescope
- Video
- Visualization
- Wireless
- X-Ray

Northeastern

GPU as a General Purpose Computing Platform

- **Speedups are impressive and ever increasing!**

Genetic Algorithm

2600 X

Real Time Elimination of Undersampling Artifacts

2300 X

Lattice-Boltzmann Method for Numerical Fluid Mechanics

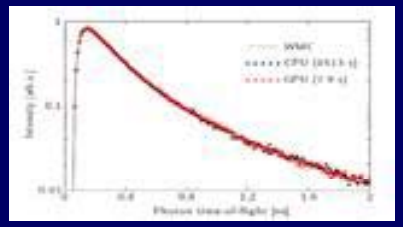
1840 X

Total Variation Modeling

1000 X

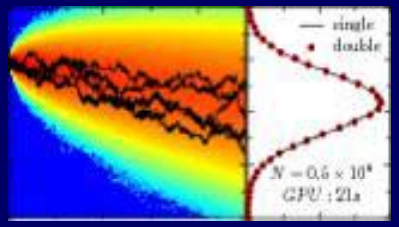
Fast Total Variation for Computer Vision

1000 X



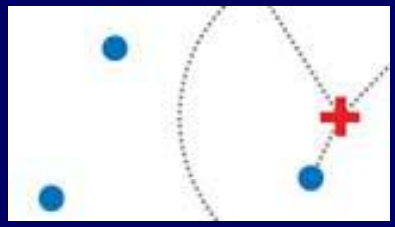
Monte Carlo Simulation Of Photon Migration

1000 X



Stochastic Differential Equations

675 X

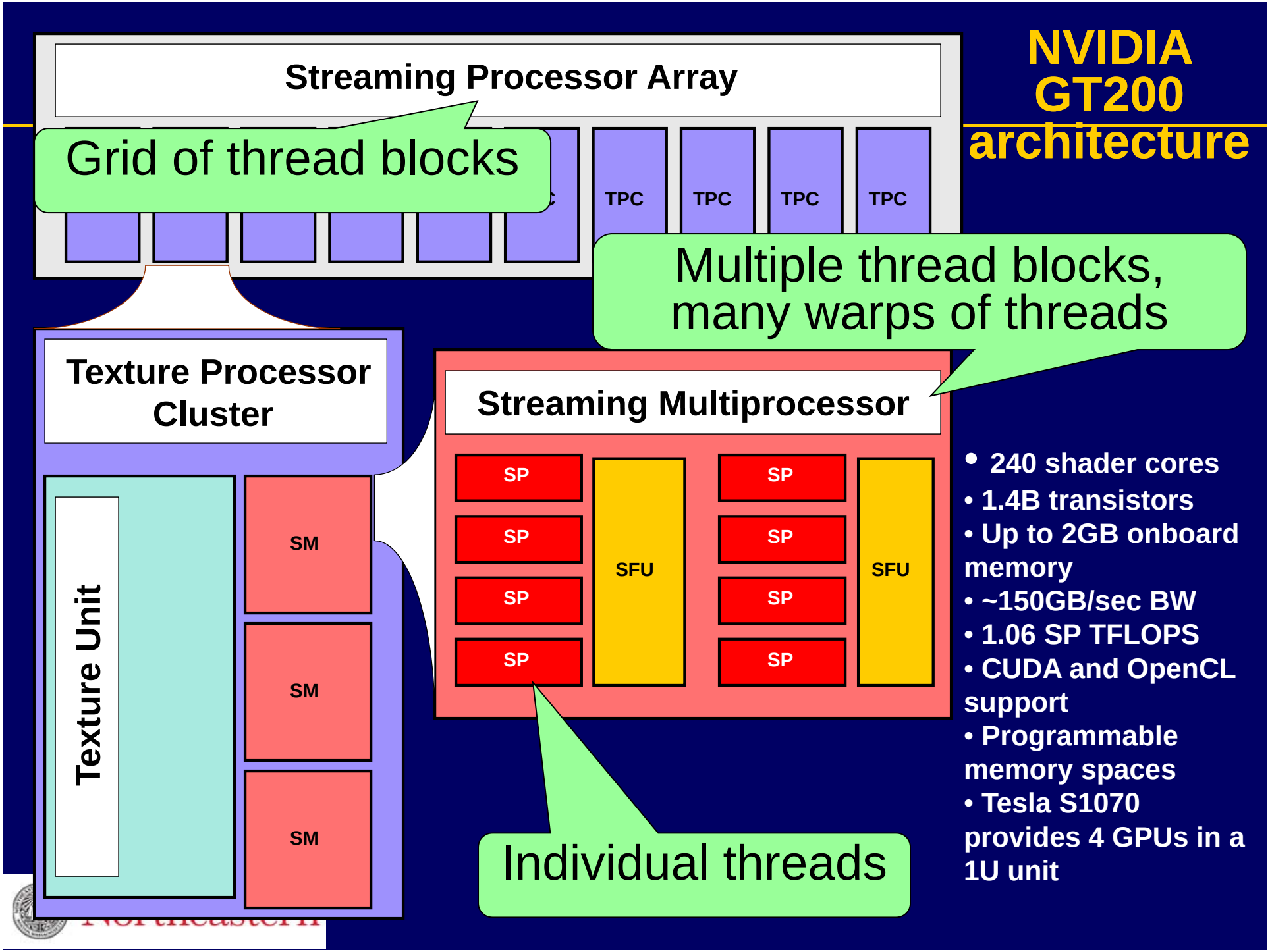


K-Nearest Neighbor Search

470 X

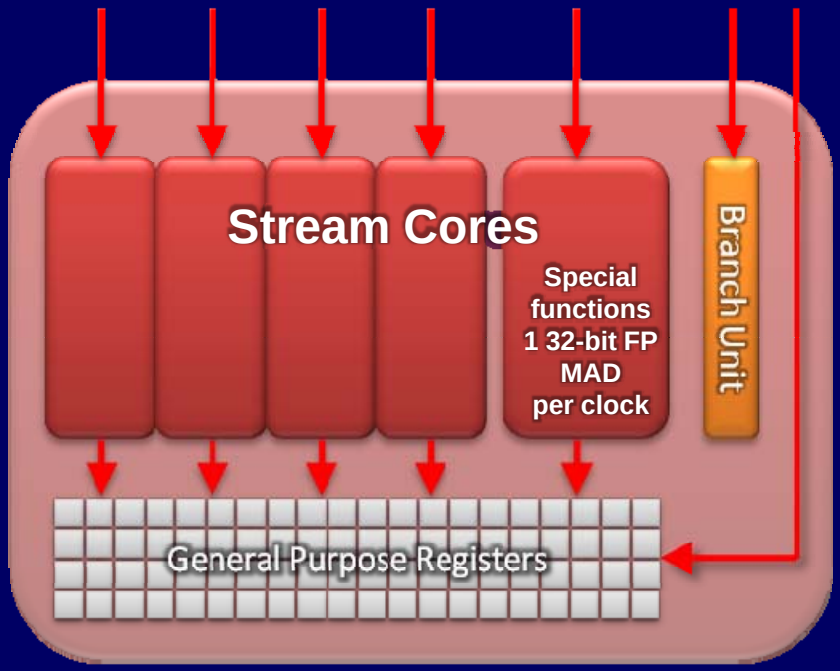
Source: CUDA Zone at www.nvidia.com/cuda/

NVIDIA GT200 architecture



- 240 shader cores
- 1.4B transistors
- Up to 2GB onboard memory
- ~150GB/sec BW
- 1.06 SP TFLOPS
- CUDA and OpenCL support
- Programmable memory spaces
- Tesla S1070 provides 4 GPUs in a 1U unit

AMD GPU Architecture

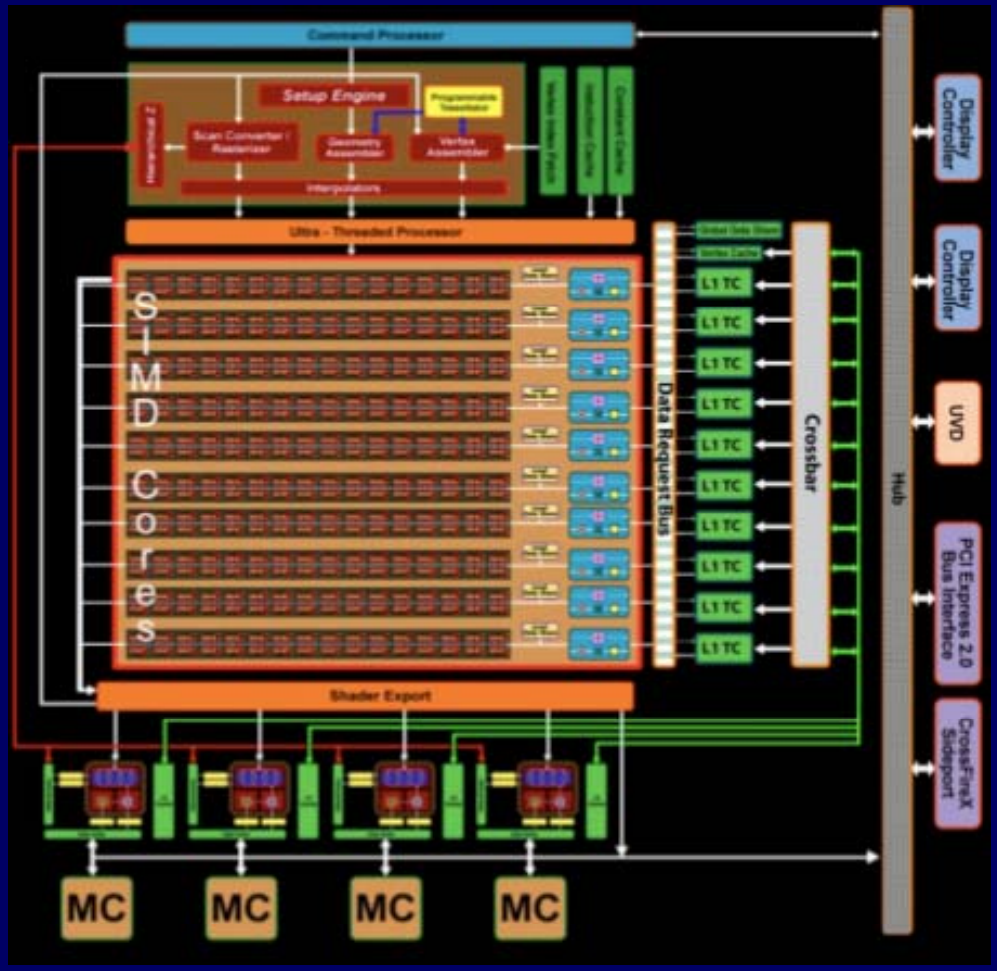


4 32-bit FP MAD per clock
2 64-bit FP MUL or ADD per clock
1 64-bit FP MAD per clock
4 24-bit Int MUL or ADD per clock

- 5-way VLIW Architecture
- 4 Stream Cores and 1 special function Stream Core
- Separate Branch Unit
- All 5 cores co-issue
- Scheduling across the cores is done by the compiler
- Each core delivers a 32-bit result per clock
- Thread processor writes 5 results per clock

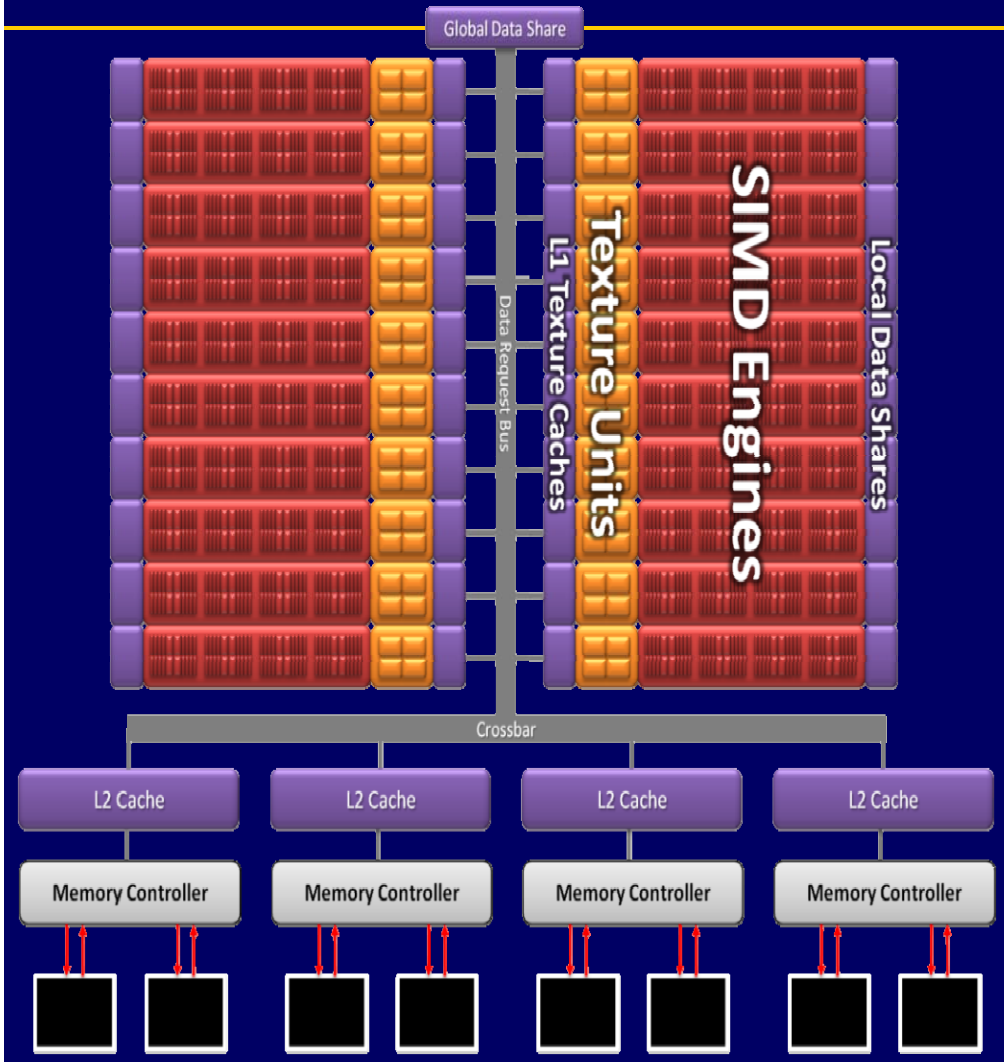
AMD/ATI Radeon HD 5870

- Codename “Evergreen”
- 20 SIMD Engines
  - 1600 SIMD cores
- L1/L2 memory architecture
- 153GB/sec memory bandwidth
- 2.72 TFLOPS SP
- OpenCL and DirectX11
- Provides for vectorized operation



Northeastern

AMD Memory System



- Distributed memory controller
- Optimized for latency hiding and memory access efficiency
- GDDR5 memory at 150GB/s
- Up to 272 billion 32-bit fetches/second
- Up to 1 TB/sec L1 texture fetch bandwidth
- Up to 435 GB/sec between L1 & L2

C for CUDA

- The language that started the GPGPU excitement
- CUDA only runs on NVIDIA GPUs
- Highest performance programming framework for NVIDIA GPUs presently
- Learning curve similar to threaded C programming
  - Large performance gains require mapping program to specific underlying architecture

Northeastern

OpenCL – The future for many-core computing

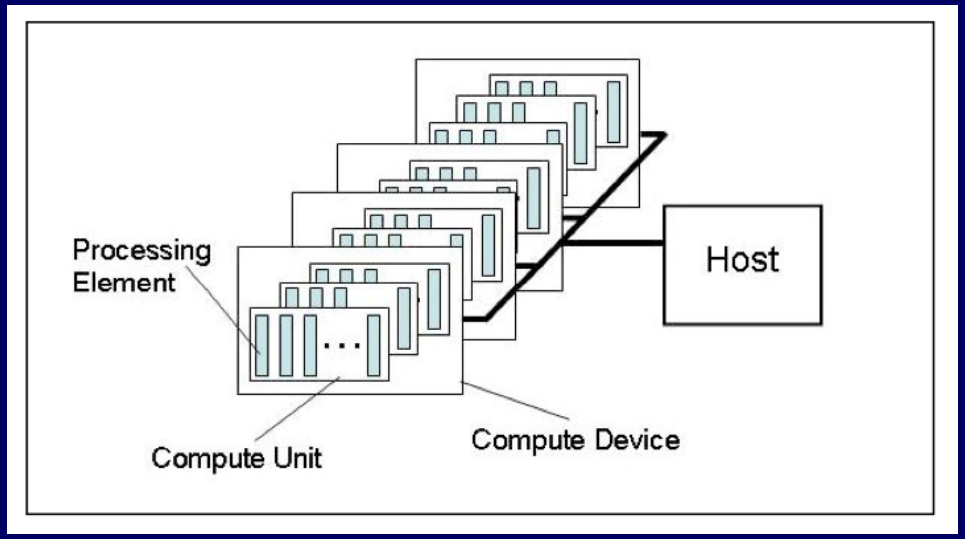
- Open Compute Language
- A framework for writing programs that execute on heterogeneous systems
- Very similar to CUDA
- Presently runs on NVIDIA GPUs and AMD multi-core CPUs/GPUs, Intel CPUs and some embedded CPUs
- Being developed by Khronos Group – a non-profit
- Modeled as four parts
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model

Northeastern

OpenCL

OpenCL Platform Model

- The model consists of a host connected to one or more OpenCL devices
- A device is divided into one or more compute units
- Compute units are divided into one or more processing elements



OpenCL

OpenCL Execution Model

- 2 main parts:
  - Host programs execute on the host
  - Kernels execute on one or more OpenCL devices
- Each instance of a kernel is called a work-item
- Work-items are organized as work-groups
- When a kernel is submitted, an index space of work-groups and work-items is defined
- Work-items can identify themselves based on their work-group ID and their local ID within the work-group

Northeastern

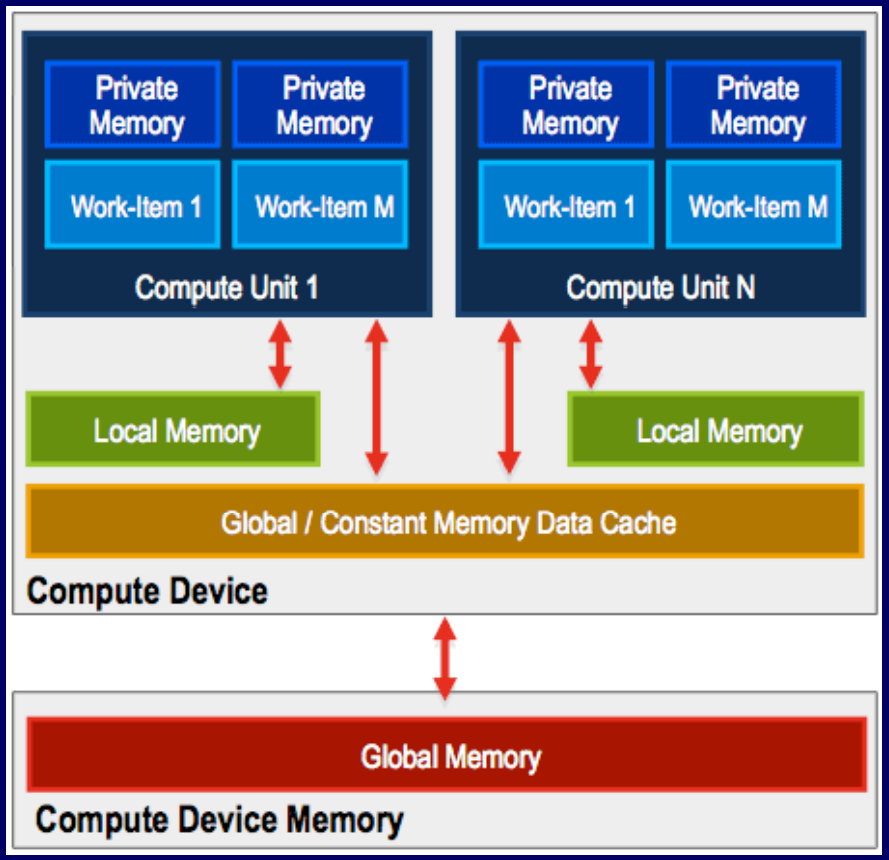
OpenCL Execution Model

- A context refers to the environment in which kernels execute
  - Devices (the elements performing the execution)
  - Program objects (the program source that implements the kernel)
  - Kernels (OpenCL functions that run on OpenCL devices)
  - Memory objects (data that can be operated on by the device)
  - Command queues are used to coordinate execution of the kernels on the devices
    - Memory commands (data transfers)
    - Synchronization
- Execution between host and device(s) is asynchronous

Northeastern

OpenCL Memory Model

- Multilevel memory exposed to programmer
- Registers (per thread)
- Local memory
  - Shared among threads in a single block
  - On-chip, small
  - As fast as registers
- Global memory
  - Kernel inputs and outputs
  - Off-chip, large
  - Uncached (use coalescing)



Northeastern

OpenCL

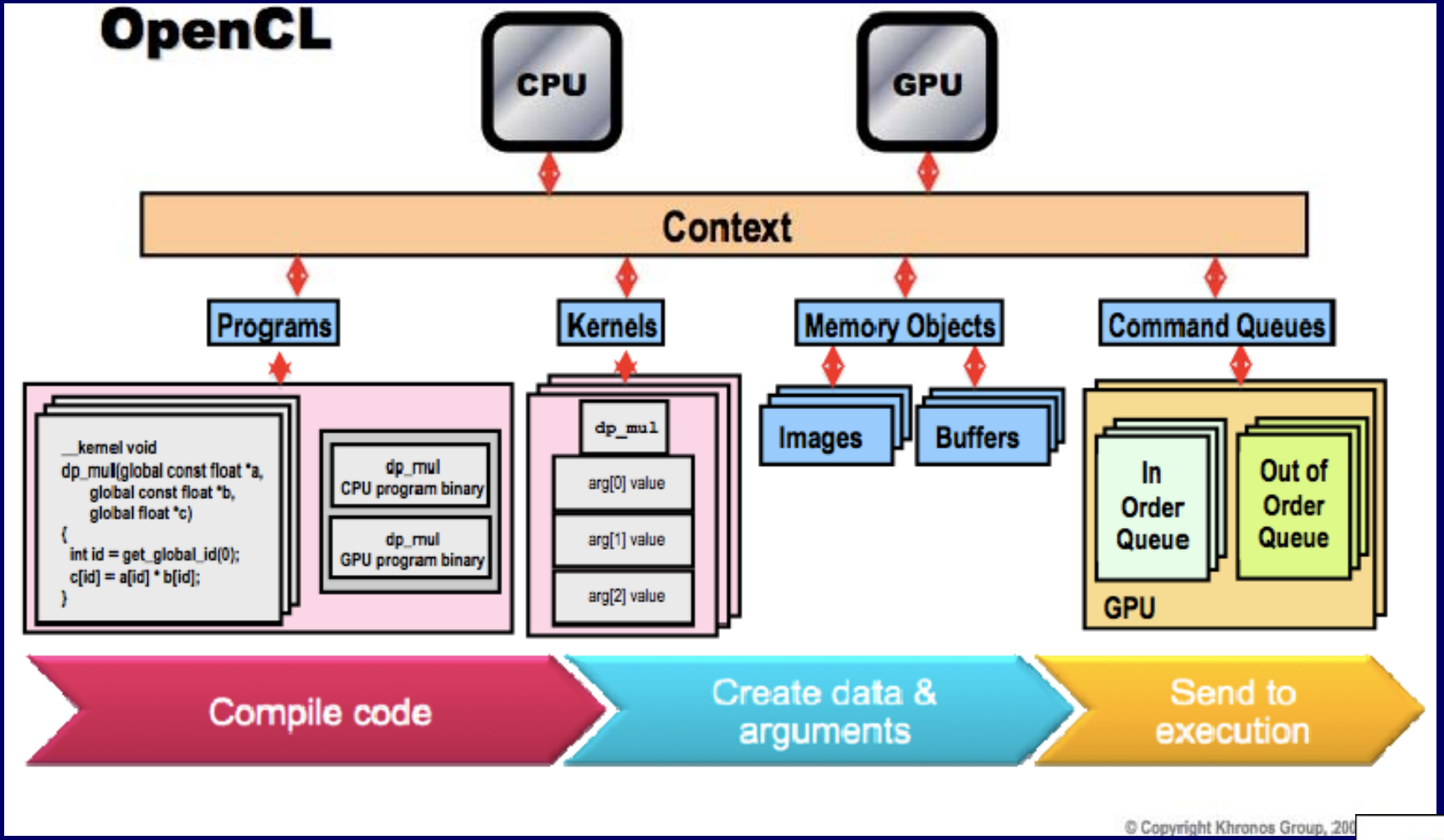
OpenCL Programming Model

- Data parallel
  - One-to-one mapping between work-items and elements in a memory object
  - Work-groups can be defined explicitly (like CUDA) or implicitly (specify the number of work-items and OpenCL creates the work-groups)
- Task parallel
  - Kernel is executed independent of an index space
  - Other ways to express parallelism: enqueueing multiple tasks, using device-specific vector types, etc.
- Synchronization
  - Possible between items in a work-group
  - Possible between commands in a context command queue

Northeastern

OpenCL

Putting it all together…..



A Look at the Future for GPUs

- AMD Fusion
  - CPU/GPU on a single chip
  - Shared-memory model
  - Reduces communication overhead
- NVIDIA Fermi
  - ECC on device memory
  - 20X speedup on atomic sync operations
  - Programmable caching

AMD Fusion – The Future for CPU/GPU Computing

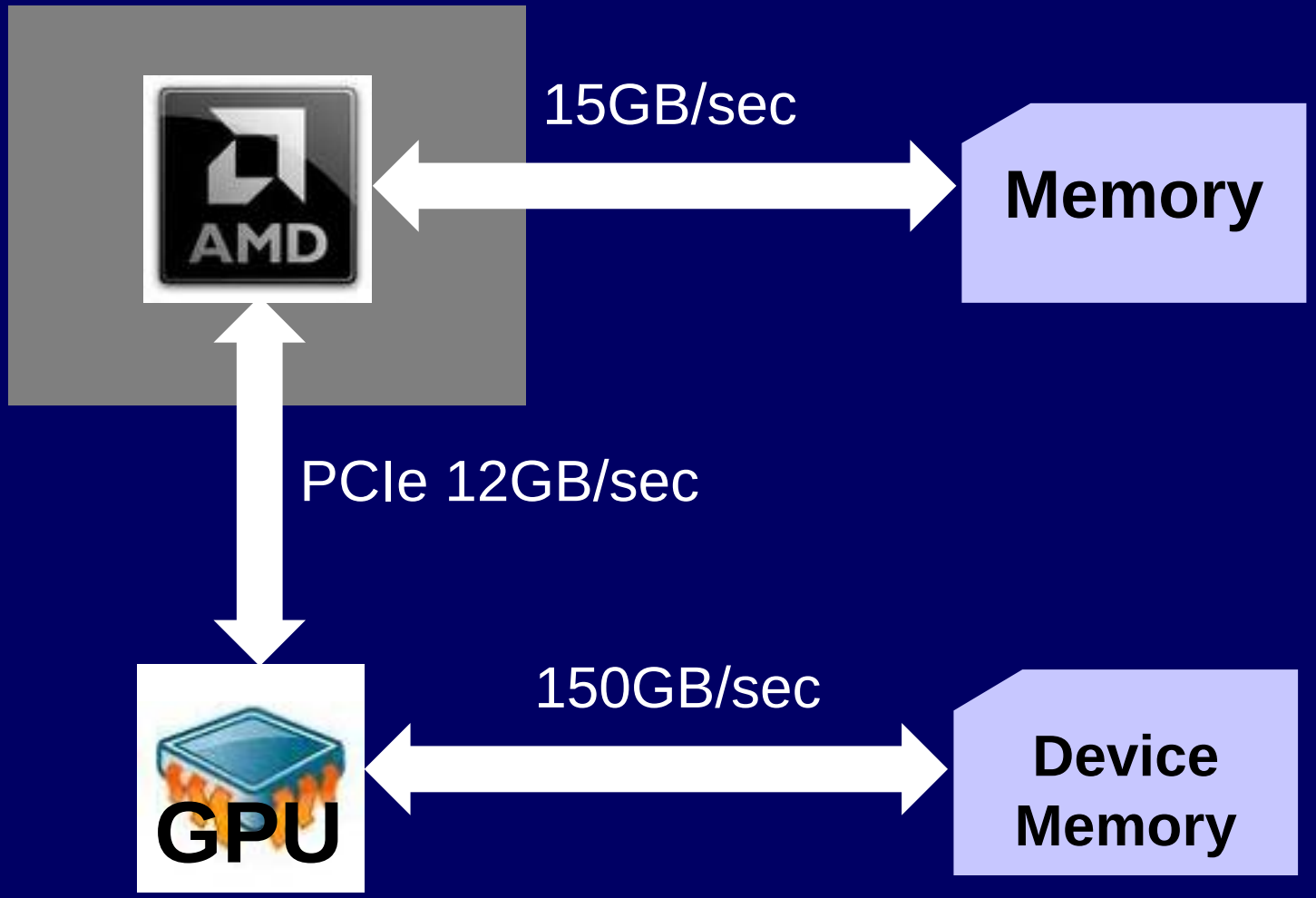
x86 CPU owns the Software World

- Windows, MacOS and Linux franchises
- Thousands of apps
- Established programming and memory model
- Mature tool chain
- Extensive backward compatibility for applications and OSs
- High barrier to entry

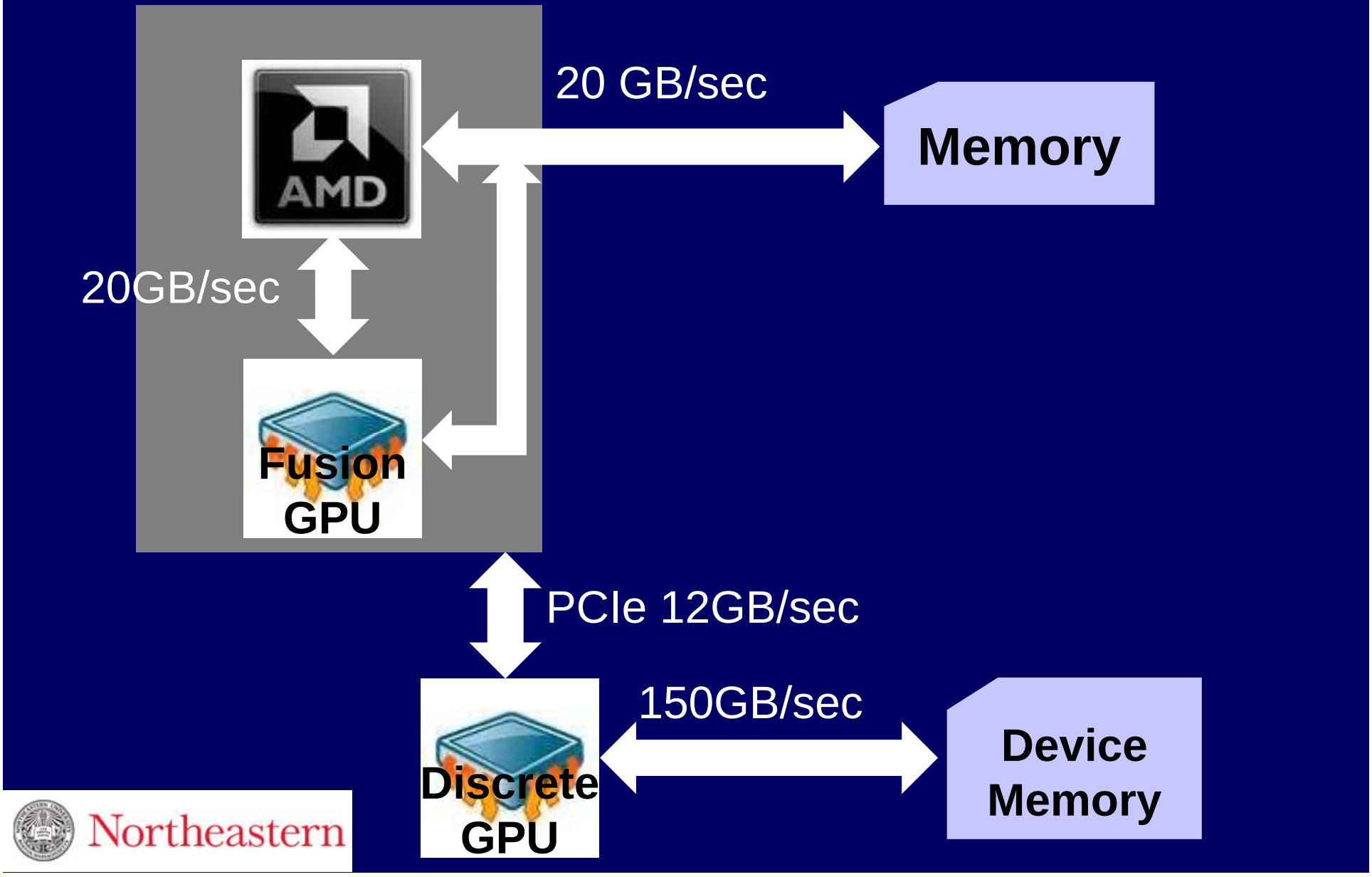
GPU Optimized for Modern Workloads

- Enormous parallel computing capacity
- Outstanding performance-per - watt-per-dollar
- Very efficient hardware threading
- SIMD architecture well matched to modern workloads: video, audio, graphics

PC with a Discrete GPU



PC with a Discrete GPU



Two New AMD Cores Tuned for Target Markets

"Bulldozer"

Performance & Scalability

Mainstream Client and Server Markets

"Bobcat"

Flexibility, Low Power & Low Cost

Low Power Markets

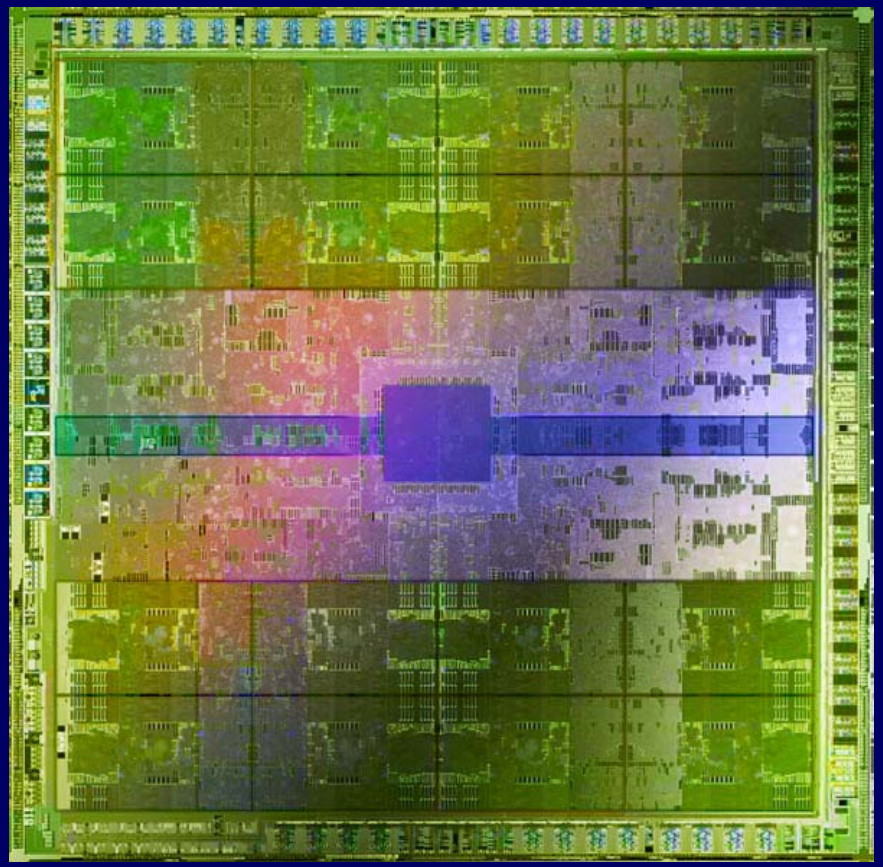
Lower Cost

Cloud Optimized

Northeastern

NVIDIA Fermi – Dedicated GPGPU device

- Fermi
  - 480 CUDA cores
  - 8X the current double precision FP performance
  - 16 Concurrent kernels
  - ECC support



Summary

- Parallel computing is here to stay
- Multi-core CPUs and many-core GPUs are everywhere
- This class first provided background in parallel thinking and computation
- pthreads parallel programming was covered was introduced
- Our next class will focus on the CUDA programming environment and NVIDIA hardware

Northeastern

NVIDIA CUDA